AFRL-RI-RS-TM-2008-22
In-House Technical Memorandum
June 2008

FORMAL METHODS TO SUPPORT THE DESIGN OF DISTRIBUTED SYSTEMS

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

NOTICE AND SIGNATURE PAGE

REPORT DOCUMENTATION PAGE	*Form Approved* **OMB No. 0704-0188**

1. REPORT DATE *(DD-MM-YYYY)* JUN 08	**2. REPORT TYPE** Final	**3. DATES COVERED** *(From - To)* Jun 07 – Sep 07

4. TITLE AND SUBTITLE FORMAL METHODS TO SUPPORT THE DESIGN OF DISTRIBUTED SYSTEMS	**5a. CONTRACT NUMBER** In-House
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Dilia E. Rodriguez	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) AFRL/RITB 525 Brooks Rd Rome NY 13441-4505	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) AFRL/RITB 525 Brooks Rd Rome NY 13441-4505	**10. SPONSOR/MONITOR'S ACRONYM(S)**
	11. SPONSORING/MONITORING AGENCY REPORT NUMBER AFRL-RI-RS-TM-2008-22

12. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. PA# WPAFB 08-3655

13. SUPPLEMENTARY NOTES
This research was funded by the 2007 RI/AFOSR Mini Grant program.

14. ABSTRACT
This work contributes to a formal framework to support the development of distributed systems: a specification serves to document a system; it can be executed to simulate the system; and it can be subjected, either directly or after some modular extension or transformation, to various kinds of formal analyses. Two on-the-fly techniques to reduce the state space were developed: one a symmetry reduction; the other a partial-order reduction. These are implemented as simple transformations of the specification of the system. A third transformational technique allows the verification of nontrivial properties not readily expressible in linear temporal logic.

15. SUBJECT TERMS
Verification, formal specification

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Dilia Rodriguez
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	UU	20	**19b. TELEPHONE NUMBER (*Include area code)*** N/A

2007 IF/AFOSR MiniGrant Report

Formal Methods to Support the Design of Distributed Systems

Dilia E. Rodríguez

AFRL / RITB
Dilia.Rodriguez@rl.af.mil
315-330-4280

1 Introduction

A distributed system is composed of different parts that work together to provide one or more functions. Its state space grows exponentially with the number of components. So it becomes extremely difficult to consider all the possible behaviors of the system, and consequently, difficult or impossible to make guarantees about it. Here we report on techniques to reduce the state space and to verify strong, nontrivial properties. This contributes to a framework to support the development of a distributed system: a specification serves to document the system; it can be executed to provide some debugging, and to simulate the system; and it can be subjected either directly or after some modular transformations to various kinds of formal analyses.

This research developed techniques that modularly transform the specification of a system to be analyzed. Two simple transformations implement on-the-fly state-space reductions, one a symmetry reduction, the other a partial-order reduction. A third transformational technique allows the verification of properties that are not readily expressible in linear temporal logic.

In designing a good system there must be a clear understanding of the properties it should satisfy. Some mathematical fields deal with structure and reasoning about properties, and these are the fields that underlie formal methods — rigorous techniques for the specification and analysis of computational systems. So formal methods are well poised to describe the structure of distributed systems, and reason about their properties.

There are a variety of formal methods for the specification and analysis of systems. They differ in the mathematics that underlie them, the kinds of properties they can specify and analyze, and the available tools associated with them. In general, there are two basic approaches to the analysis and verification of properties: theorem proving and state-space exploration. Of the two approaches, theorem proving is, in theory, applicable to systems with state spaces of any size, but requires a high level of expertise; while a state-space exploration approach is applicable only to finite-state systems, but requires less expertise, and is more automatic. Both formal approaches require more powerful techniques to provide practical support for the development of distributed systems.

The formal system selected for this research is Maude — an executable logic of concurrent change. It has been shown to be a framework for a wide range of computational models and logics. This means not only that models and properties expressible in these formal systems can be expressed and analyzed in Maude, but that the *sound* integration of various such descriptions becomes possible, and with this more powerful means of studying systems. Furthermore, Maude provides various formal methods to study a specification as it is being developed. *Executable specifications* can serve to debug the specification, and provide prototypes of the specified system. A *search* command serves to explore the state space, making it possible to check the existence of different scenarios in the system, as well as checking invariant properties of the system. *Model checking* makes possible the verification of properties expressible in linear temporal logic. In isolation and in combination these methods can be used to study a specification as it is being developed. The main objective of this research was to develop techniques that would allow the application of these methods to specifications of systems of larger size and more complex properties.

To motivate and evaluate the techniques developed, a client-server protocol was studied, a simplification of the Chain-Replication protocol developed by van Renesse and Schneider [7]. Three techniques were developed: an on-the-fly symmetry reduction technique that modularly augments a specification, a technique with auxiliary data to support the verification of strong properties, and a technique to reduce the state space using a partial-order approach.

All these techniques, though only applied to a singe protocol in this study, are general techniques that could be applied to a wide range of protocols. The symmetry-reduction technique can be applied to any system with identical components. This technique was very effective in reducing the state space. While some techniques are very effective in reducing the state space when verifying shallow properties, verification of strong properties is a greater challenge. The second technique permitted the verification of the strong property the Chain-Replication protocol should satisfy, and should be applicable to the verification of strong properties of other protocols. Finally, the partial-order reduction technique developed resulted in a small reduction of the state space when applied to the client server protocol under study. The effectiveness of this technique, however, is determined by the interdependencies of the transitions of the protocol. So further evaluation of this technique should include its application to other protocols. A significant innovation of this technique is that it is applied to the specification of a system, rather than to the model checking algorithm that will be used to verify properties of the specification, which is the usual case. This makes possible the exploitation of properties of the specification, that might not be possible when the technique is applied to the model checker. Thus there should be further investigation, optimization and evaluation of this technique. I will be presenting a paper describing this research at the 7^{th} International Workshop on Rewriting Logic and its Applications, WRLA08.

Section 2 introduces Maude preliminaries, in particular those related to object-based specifications. The specification of the simplication of the Chain-Replication protocol is presented in Section 3. The next section discusses the on-the-fly symmetry reduction technique and its application to the protocol. Section 5 motivates and describes a technique that uses auxiliary data to verify the strong-consistency property this protocol should satisfy. Section 6 describes a partial-order reduction technique that modifies the specification of the protocol. The conclusions and future work are presented in Section 7.

2 Maude Preliminaries

Maude [2][3] is an executable language based on rewriting logic [5], a logic of concurrent change. In rewriting logic, a concurrent system is specified by a rewrite theory $\mathcal{R} = (\Sigma, E, R)$, where (Σ, E) is an equational theory with the signature Σ specifying sorts (types) and operations; E, a set of equations on Σ-terms; and R, a set of labelled conditional rewrite rules, of the form $l : t \longrightarrow t'$ if *cond*. The equational theory describes the distributed structure of the system, while the conditional rewrite rules define its basic concurrent transitions.

A rewrite theory corresponds to a system module in Maude. For system modules that satisfy some executability requirements [3], a rewrite rule describes not only a transition between states, but a transition between equivalence classes of states, where the equivalence classes are defined by the equations of the rewrite theory. Thus, properly defined executable specifications in Maude have this condensed and reduced state space.

Maude supports object-based models, with a predefined module declaring sorts for the essential concepts, namely `Object`, `Msg`, and `Configuration`.

```
mod CONFIGURATION is sorts Object Msg Configuration .    ...
```

A configuration is a multiset of messages and objects. In particular, a single message or a single object is a configuration. Maude supports subsorts, so this can be expressed as follows:

```
subsort Object Msg < Configuration .
```

A configuration is described by a term of sort `Configuration` constructed with the following operators:

```
op none : -> Configuration [ctor] .
op __ : Configuration Configuraton -> Configuration
        [ctor config assoc comm id: none] .
```

The first takes no arguments, and represents a configuration with neither objects nor messages. The second takes two arguments, which are juxtaposed (the `_` is a placeholder, and there is no syntax between the arguments), and is declared with attributes of a multiset: it is associative, commutative, and has identity `none`.

A typical configuration has the form $O_1 \ldots O_n \; M_1 \ldots M_m$, where the O's represent objects and the M's messages. The most general form of a conditional rewrite rule for an object-based model is of the form:

$$l : M_1 \ldots M_m \; O_1 \ldots O_n \to O'_{i_1} \ldots O'_{i_k} \;\; Q_1 \ldots Q_p \;\; M_1 \ldots M_q \;\; \textit{if} \; C$$

This rule, labelled by l, represents transitions in which, if the condition C holds for the configuration on the left side of the rule, messages $M_1 \ldots M_m$ are consumed; the states of some of the objects $O_1 \ldots O_n$ change, becoming $O'_{i_1} \ldots O'_{i_k}$, $k \leq n$, with the rest disappearing; and new objects $Q_1 \ldots Q_p$ and messages $M_1 \ldots M_q$ are created.

3 A Client-Server Protocol

The client-server protocol studied is a simplified version of the Chain-Replication protocol developed by van Renesse and Schneider [7]. Their protocol has m servers and n clients, but from the perspective of a client there is a single server. The innovation of the protocol is in achieving fault tolerance and high throughput through the collective service provided by the servers, but the state-explosion problem is present in configurations with a single server and several clients. The simplified version of the Chain Replication protocol is used to develop techniques to reduce the state space, and to define (and check in its limiting case) the property the Chain-Replication protocol should satisfy.

In this protocol the server stores an object, whose value a client may observe by making queries, or change by requesting updates. Informally, the property this protocol must satisfy is that any response to a query by a client must reflect prior updates.

An object-based model of this protocol has servers and clients as objects, and requests and replies as messages.

```
sorts Client Server .    subsorts Client Server < Object .
sorts Request Reply .    subsorts Request Reply < Msg .
```

A client is represented using the following operator:

```
op < client_ | request-count :_,  outstanding :_,  value :_>
   : NzNat Nat Bool Value -> Client [ctor] .
```

A nonzero natural serves to identify a client, and a request count is used to limit the number of requests a client can make, ensuring that the state space remains finite. Each client keeps the value of the object, as it has observed it through requests to the server. The protocol stipulates that a client may have at most one outstanding request. For the purposes of this study it is not useful to consider failures or messages lost, and so as long as there is an outstanding request the client may not issue another. Boolean attribute `outstanding` indicates whether the client has initiated a request for which it is expecting a reply.

A request instructs the server to perform an operation on the object: a query is a read operation, while an update is a write operation.

```
    op query[_] : NzNat -> Request [ctor] .
    op update[_:_] : NzNat Value -> Request [ctor] .
```

The action of a client sending a query is represented by the following conditional rule.

```
crl [send-query]
    < client N | request-count : K,   outstanding : false,  value : V >
 => < client N | request-count : s K, outstanding : true,   value : V >
    query[N]    if K < lim .
```

Similarly, a client sending an update is represented by the following rule:

```
crl [send-update]
    < client N | request-count : K,   outstanding : false,  value : V >
 => < client N | request-count : s K, outstanding : true,   value : V >
    update[N : val(N, s K)]    if K < lim .
```

Either request may be made only if there is no outstanding request, that is, if `outstanding` is false. If the request is a query, it is represented symbolically by `query[N]`, which identifies the requesting client. If the request is an update, it must include the value the client is submitting. This is represented symbolically as `val(N, K)`, which indicates that this is the value client `N` submitted in its `s K`-th request.

How requests or replies are transported from sender to receiver is not determined by the protocol, and so a term of sort `Msg` in the configuration represents a message that has been sent but not received.

The server receives and processes requests. It is represented using the following operator:

```
    op < server | pending :_,  value :_>
       : RequestQueue Value -> Server [ctor] .
```

It receives a request by removing it from the configuration and enqueueing it in the `pending` queue.

```
    rl [get-request]
        < server | pending : Q,     value : V >  R
     => < server | pending : Q ; R, value : V > .
```

As the server processes a request of a client, it sends a reply confirming the operation. It replies to a query with the current value of the object; and to an update with the value the client had requested be assigned to the object, which is now the current value. So a reply has the following syntax.

```
    op reply-to[_:_] : NzNat Value -> Reply [ctor] .
```

where the first argument identifies the client to which it is addressed.

The act of the server processing a request and replying is represented by a single rule. Processing a query preserves the value of the object.

```
    rl [respond-to-query]
        < server | value : V, pending : query[N] ; Q >
     => < server | value : V, pending : Q >   reply-to[N : V] .
```

Processing an update may change it.

```
    rl [respond-to-update]
        < server | value : V', pending : update[N : V] ; Q >
     => < server | value : V,  pending : Q >   reply-to[N : V] .
```

A reply in the configuration represents a message in transit from the server to some client. A client receives a reply by the following rule.

```
    rl  [get-reply] :
        reply-to[ N : V' ]
        < client N | request-count : K, outstanding : true, value : V >
     => < client N | request-count : K, outstanding : false, value : V' > .
```

The attribute `outstanding` becomes false, since receiving the reply concludes the operation.

Thus, the state of this system consists of one server, one or more clients, and possibly various requests and replies. This configuration is enclosed within delimeters as follows:

```
    sort TConfiguration .
    op {_} : Configuration -> TConfiguration [ctor] .
```

representing the state as a term of sort `TConfiguration`.

To analyze a protocol using methods that explore the state space requires that the protocol be instantiated. Two parameters characterize an instantiation of the client-server protocol just described: `size`, the number of clients; and `lim`, the number of requests a client may make. The server and all clients are initialized with a special value, and in all experiments `lim` was 3.

The `search` command, which is part of the Maude system, allows one to explore the state space in a variety of ways (see [3]). Through arguments and various forms it may return all states (for finite state spaces), or all states satisfying some property, or the first n states it finds, for a specified n. The result of the command includes the number of states examined in obtaining the result. Table 1 shows the results of various experiments in which the final states of a given instantiation were requested, using the following command:

```
    search { init(size) } =>! TC:TConfiguration .
```

For the instantiation of size 4, that is, with one server and four clients, after much swapping the computation would abort. These results show that even for small instantiations of a very simple protocol state spaces can be very large. The next section describes a way of reducing the state space.

size	2	3	4
total states	4,933	952,747	$> 4,194,304$
final states	37	511	
cpu time (ms)	290	35,978	628,583
real time (ms)	2,454	45,009	22,307,029

Table 1: Metrics for basic specification.

4 Symmetry Reduction

Many distributed systems include identical components. Thus, if one such component reaches a particular state in one of the possible behaviors of the system, an identical component would reach the same state in a similar behavior. Exploiting symmetry as a general approach is not new. When and how it is exploited can result in different techniques and effectiveness. This section first presents mathematical preliminaries for exploiting symmetry. (For a more complete presentation see [1][4].) Then it describes a class of systems and how to use symmetry to reduce the state spaces of these systems.

4.1 Mathematical Preliminaries

A transition system is a pair $\mathcal{A} = (A, \rightarrow_{\mathcal{A}})$ where A is a set of states and $\rightarrow_{\mathcal{A}} \subseteq A \times A$ is a binary relation called the transition relation. A permutation π on a finite set A is a function $\pi : A \rightarrow A$ that is one-to-one and onto. It is an automorphism on $\mathcal{A}$ if it is such that for all $a, a' \in A$, $a \rightarrow_{\mathcal{A}} a'$ if and only if $\pi a \rightarrow_{\mathcal{A}} \pi a'$. Given an automorphism π on $\mathcal{A}$, $a_0, \ldots a_n$ is a path in $\mathcal{A}$ if and only if $\pi a_0, \ldots \pi a_n$ is a path in $\mathcal{A}$. Any set of automorphisms on $\mathcal{A}$ closed under composition and the inverse operation is a group. An automorphism group G on $\mathcal{A}$ induces a relation $\simeq_G$: $A \times A$ such that $a \simeq_G a'$ if and only if there exists an automorphism $\pi \in G$ such that $a = \pi a'$. A congruence on $\mathcal{A}$ is an equivalence relation $\approx$ where for all $a_1, a_2 \in A$ such that $a_1 \approx a_2$, if there exists $a_1' \in A$ such that $a_1 \rightarrow_{\mathcal{A}} a_1'$, then there is $a_2' \in A$ such that $a_1' \approx a_2'$ and $a_2 \rightarrow_{\mathcal{A}} a_2'$. The relation $\simeq_G$ is a congruence on $\mathcal{A}$.

Let $[a]$ denote the class of states equivalent to a. For any $a, a' \in A$, if $a \rightarrow_{\mathcal{A}} a'$, then for all $a_1 \in [a]$ there exists $a_1' \in [a']$ such that $a_1 \rightarrow_{\mathcal{A}} a_1'$. A quotient transition system $\mathcal{A}_G = (A_G, \rightarrow_{\mathcal{A}_G})$ of transition system $\mathcal{A}$ with respect to a permutation group G is defined by $A_G = \{[a] \mid a \in A\}$ and $\rightarrow_{\mathcal{A}_G} = \{[a] \rightarrow_{\mathcal{A}_G} [a'] \mid a \rightarrow_{\mathcal{A}} a'\}$. Given an equivalence class $[a]$ in $\mathcal{A}_G$, some $*a \in [a]$ may be chosen to represent $[a]$, and a quotient representative system $\mathcal{A}_{G^*} = (A_{G^*}, \rightarrow_{\mathcal{A}_{G^*}})$ of may be defined by $A_{G^*} = \{*a \mid [a] \in A_G\}$ and $\rightarrow_{\mathcal{A}_{G^*}} = \{*a \rightarrow_{\mathcal{A}_{G^*}} *a' \mid [a] \rightarrow_{\mathcal{A}_G} [a']\}$. Then$a$ is reachable from a_0 in $\mathcal{A}$ if and only if $*a$ is reachable from $*a_0$ in $\mathcal{A}_{G^*}$.

4.2 States, Indexed Objects and Automorphisms

In the client-server protocol, clients are specified uniformly as indexed objects, and indices are further used to represent values symbolically. Thus, the states of the system may be expressed as functions on indices. More generally, a state with a finite set of identical components may be described using a finite set of indices I, and a function $s : I \to A$, where A is a set of states. A permutation $\pi_I : I \to I$ induces a permutation $\pi : A \to A$ defined by $\pi\, s(i_1, \ldots, i_n) = s(\pi_I i_1, \ldots, \pi_I i_n)$. The question now is which permutations on states induced by permutations on indices are automorphisms.

To determine this, consider the effect of permuting indices on each of the rules of the specification. A close examination of all the rules of the client-server protocol shows that whether a rule is enabled is independent of the values of indices. Thus, for the client-server protocol all permutations on indices induce permutations on states that are automorphisms. This will hold for any specification in which objects with identical behavior are specified uniformly using indices. Two objects have identical behavior when their set of possible states is the same, and whenever they have the same state, they react in the same way. Next, the fact that all permutations of indices induce automorphisms on states is exploited in a technique for reducing the state space on the fly.

4.3 On-the-fly State-Space Reduction

To analyze a system using state-space exploration methods the parameters of the specification must be instantiated. In particular, a system with identical components must be instantiated with a fixed number of such components, using indices to differentiate among them. As seen above, the set of all permutations of these indices induces a group $\mathcal{G}$ of permutations on states that are automorphisms. The equivalence class with respect to $\mathcal{G}$ for state s is $[s] = \{\pi s \mid \pi \in \mathcal{G}\}$, and some $*s \in [s]$ is selected as its representative. We consider how to compute with representatives of these equivalence classes.

There is no guarantee that transitions map representative states to representative states. So there is the need to recognize when a transition has occurred, and to obtain the representative corresponding to the state resulting from the transition.

From a specification $\mathcal{R}$, we construct in a modular way one that implements on-the-fly state-space reduction, $\mathcal{R}^* = (\Sigma \cup \Sigma^*, E \cup E^*, R')$. Signature Σ^* and equations E^* define permutations, how to apply them, construct equivalence classes, and chose representative states, and the set of rules R' is a trivial transformation of R.

New syntax and equations in (Σ^*, E^*) define permutations on indices.

```
sorts Replacement Perm .  subsort Replacement < Perm .
op _|->_ : NzNat NzNat -> Replacement [ctor] .
op __ : Perm Perm -> Perm [ctor assoc] .
```

The application of a permutation `P` to a term `t` is expressed as `P >> t`. We do not present the definition of permutations on indices, but indicate that `all-perms N` is the group of all permutations on $\{1, \ldots, N\}$.

For the client-server protocol the permutation on states is specified by equations effecting the change of indices throughout a term representing a state. For example, the effect of such a permutation on a symbolic value and on a reply is given by:

```
    eq P >> val(I, K) = val(P >> I, K) .
    eq P >> reply-to[I : V] = reply-to[P >> I : P >> V] .
```

while the effect on clients and servers is as follows:

```
eq P >> < client I | request-count : K, outstanding : B, value : V >
  = < client (P >> I) | request-count : K, outstanding : B, value : P >> V > .
eq P >> < server | pending : Q, value : V >
      = < server | pending : P >> Q, value : P >> V > .
```

The augmentation (Σ^*, E^*) to the original specification $\mathcal{R} = (\Sigma, E, R)$ also defines how to select a representative among the states in $[s]$. This is defined using a predefined parametric predicate on terms that Maude provides. Maude is a reflective language. Every term, equation and rule can be represented at a metalevel. At this level Maude defines a total order on the metarepresentations of terms. This total order can be checked from the ground or object level with the predicate `lt`. Thus, if t_1 and t_2 are terms at the object level, they have some metarepresentations T_1 and T_2 at the metalevel. Because there is a total order on metarepresentations of terms, for distinct T_1 and T_2, the predicate `lt`(t_1, t_2) is defined to be true if and only if T_1 precedes T_2. This predicate is used to select a representative for an equivalence class $[s]$, where s is a term representing a state of the system. The representative is chosen to be the state whose metarepresentation is least among the metarepresentations of all the states in $[s]$.

Let s be any state of the system specified by $\mathcal{R} = (\Sigma, E, R)$. If the system is object-based, state s is a term of sort `Configuration`. Introduce the following operators:

```
    op all-perms : -> PermSet .
    op _|_ : PermSet Configuration -> Configuration [frozen (1 2)] .
```

The constant `all-perms` is the group of all permutations on the set of indices $\{1, \ldots, n\}$, where n is the number of identical components in the instantiated system. The operator `_|_` applies a set of permutations to a state, and returns the representative of the resulting set of states. This operator takes two arguments, and its attribute `frozen (1, 2)` prohibits the application of rules to either argument. The selection of the representative of $[s]$, for a configuration state s is defined for `Configuration C`, `Perm P`, and `NePermSet sP` as follows:

```
    eq emptyPermSet | C = C .
    eq P | C = emptyPermSet | if lt(P >> C, C) then P >> C else C fi .
    eq P U sP | C = sP | if lt(P >> C, C) then P >> C else C fi .
```

size	2	3
total states	2,473	176,897
reduction	50%	81%
final states	19	95
cpu time (ms)	3,089	821,392
real time (ms)	4,861	911,346

Table 2: Metrics for basic specification with on-the-fly symmetry reduction.

Recall that the goal is to compute with representatives of the equivalence classes of a quotient system. Since transitions on representative states do not necessarily reach representative states, whenever a transition leads to a state s, this state must be mapped to the representative of $[s]$, that is, to $*s$.

The next question is how to detect that a transition has occurred. For this introduce a subsort `Marker` of `Msg`, with constants `?` and `!`. Modify each rule in R by adding `?` to its left side, and `!` to its right side. Given an initial state with a single pretransition marker `?`, the consumption of `?` and appearance of `!` indicates that a transition has occurred. Then the representative state can be selected before reintroducing `?`.

```
eq  { ! C } = { ? (all-perms | C) } .
```

Table 2 shows the results obtained when using the on-the-fly state-space reduction just described.

5 Strong Consistency

The protocol described in Section 3 is a simplification of the Chain-Replication protocol described in [7]. The property that should hold for this protocol should hold also for its simplification. It requires that query and update operations be executed in some sequential order, and that the effects of update operations be reflected in the results returned by subsequent query operations. Thus, the correctness of this protocol requires that the value a server has at one time agree with the value a client will receive at a later time. The protocol presented in Section 3 does not permit the verification of this property. It is a minimal specification, in which the state is as simple as can be to describe the protocol. The strong-consistency property the protocol should satisfiy requires a state with more information. Information that is specific to each client. This section transforms the specification presented in Section 3 into one that will support the verification of the strong-consistency property of the client-server protocol.

An important form of property in linear temporal logic is

$$\Box(\phi \rightarrow (\Diamond\psi)).$$

This means that for any path, whenever a state satisfies property ϕ there will be some later state in the path that will satisfy property ψ. The form of the property the client-server protocol should satisfy, however, is of the form:

$$\forall X. \forall\, i. \Box(\phi_i(X) \rightarrow (\Diamond \psi_i(X))).$$

Here X is a value the server assigns to the object, and i identifies a client. The predicate $\phi_i(X)$ states that the server replies to client i with value X; while predicate $\psi_i(X)$ states that client i receives X in a reply. No such binding of the variable X, however, is expressible in linear temporal logic.

In fact, the property the client-server protocol should satisfy in all states is concerned not only with the eventual value the client will receive, but also with the value it currently has. Thus, the form of the property is more complex than the one described above, and remains not directly expressible in linear temporal logic. So we transform the specification of Section 3 to be able to verify this property.

That specification does not keep any information about the required agreement between server and client. This will now be added. The server is the keeper of the value of the object; while a client may request update and query operations. These are not instantaneous, so we define what it means for a client and server to agree on the value of the object.

This protocol allows a client to have at most one outstanding request for an operation. A client initiates the operation by sending a request, marked by the attribute `outstanding` becoming true. The server eventually receives it, processes it, and replies to the client. When the client receives the response its `outstanding` attribute becomes false, and the operation is completed.

With the reception of the reply the client updates its value of the object. This should result in the client agreeing with the value the server had when it processed the last request by this client. This is the condition that should hold whenever the `outstanding` attribute has value false.

There are three stages while a request is outstanding. The first begins when the request is sent (with `send-query` or `send-update`). The request becomes part of the configuration. The second begins when the `get-request` rule removes this request from the configuration and enqueues it in the `pending` attribute of the server. During these two stages the agreement should still be that the client should have the same value of the object as the server had when the server processed the last request by this client.

The last stage begins when the server processes the request, which may be with the `respond-to-query` or `respond-to-update` rule, and sends the reply. This reply to the client becomes part of the configuration. The agreement condition during this stage is that the client should have the value the server had when it processed the previous to last request by this client, or if this is the first request by this client, the client should have its initial value of the object.

To be able to determine whether the required agreement holds at all times the specification will have auxiliary data.

```
sort AuxData .    subsort AuxData < Msg .
op (_[_]_) : Value NzNat Value -> AuxData .
```

size	2	3
total states	9,025	3,253,621
cpu time (ms)	1195	668,728
real time (ms)	1212	7,046,545

Table 3: Metrics for specification that supports verification of strong consistency.

For each client, the values the server had when it processed the last and previous to last requests will be kept: `(P [I] L)`.

The only other change to the original specification is to the rules that process the requested operations:

```
rl  [respond-to-query] :
    (P [N] V') < server | value : V, pending : query[ N ] ; Q >
 => (V' [N] V) < server | value : V, pending : Q  >  reply-to[N : V] .

rl  [respond-to-update] :
    (P [N] V') < server | value : V',pending : update[ N : V] ; Q >
 => (V' [N] V) < server | value : V, pending : Q >  reply-to[ N : V ] .
```

which now must update the auxiliary data to reflect the value the server had when it processed the last and previous to last operations requested by this particular client.

So in all states a client should have one of the last two values the server had when processing a request by this client. During the third phase of an outstanding request by client `I` the configuration (i.e. state) includes `reply-to[I : V]` as well as the auxiliary datum `(P [I] V)`. In any state during this stage client `I` should have value `P`. Otherwise, when there is no reply for client `I`, it should have the last value in `(P [I] V)`, that is, `V`.

The search command can be used to verify that this property holds for all clients in all states. Simply search for any state that satisfies the negation of the required property. The following search command seeks states that violate the agreement between server and client that was described above.

```
search { init(size) } =>*
{ < client I:NzNat |
      request-count : K:Nat, outstanding : B:Bool, value : V':Value >
  ( P:Value [I:NzNat] V:Value )  C:Configuration }
such that
   ( (reply-to[I:NzNat : V:Value] in C:Configuration)
     and (V':Value =/= P:Value) )
or ( (not (reply-to[I:NzNat : V:Value] in C:Configuration) )
     and (V':Value =/= V:Value) ) .
```

If no such state is found then the instantiation of the protocol that was subjected to this search satisfies the strong consistency property.

Instantiations of the specification with one server and two clients, and with one server and three clients, were found to be strongly consistent. Table 3 shows the results of the experiments that obtained these verifications.

size	2
total states	4519
final states	90
cpu time (ms)	5357
real time (ms)	9053
property verification	
cpu time (ms)	5046
real time (ms)	5052

Table 4: Metrics for symmetry reduction in the verification of strong consistency.

Table 4 gives the metrics for the search of all final states, and for the verification of consistency using symmetry reduction.

6 Partial-Order Reduction

Section 4 describes how to compute with representative states. This section describes how to compute representative paths. Having introduced the property the client-server should satisfy it becomes possible to consider partial-order reduction (POR). This approach is based on properties that some transitions have: for some pairs of transitions the order in which they are executed does not matter, and some transitions do not affect whether a property holds. Usually POR techniques are combined with model-checking algorithms. Here, instead, the POR approach is used to modify the specification of the system.

The main point about the POR approach is that in exploring the state space some transitions are ignored. A finite state transition system is a tuple (S, S_0, T, AP, L), where S is a finite set of states, $S_0 \subseteq S$ is a set of initial states, T is a finite set of transitions, such that each $\alpha \in T$ is a partial function $\alpha : S \to S$, AP is a finite set of propositions, and $L : S \to 2^{AP}$ is a labelling function. The set of enabled transitions in state s is denoted by $\mathit{enabled}(s)$. If $\alpha \in \mathit{enabled}(s)$, the state reached by taking α is denoted by $\alpha(s)$. POR techniques explore only transitions in some set $\mathit{ample}(s) \subseteq \mathit{enabled}(s)$.

Two concepts are used to define such a set. First, an independence relation $I \subseteq T \times T$ that is symmetric and antireflexive. Transitions $\alpha, \beta \in I$ if for all states $s \in S$, if $\alpha \in \mathit{enabled}(s)$, then $\beta \in \mathit{enabled}(s)$ if and only if $\beta \in \mathit{enabled}(\alpha(s))$. Second, a transition α is invisible if for all $s \in S$, $L(s) = L(\alpha(s))$. Based on these concepts [1] gives four conditions that $\mathit{ample}(s)$ must satisfy in order to preserve the satisfaction of properties invariant under stuttering.

Consider these concepts for the client-server protocol. In Maude transitions are specified by rules, usually having variables. Partially instantiating the rules of the client-server protocol the following transitions are used to define the independence relation: q_i, client i makes a query; u_i, client i requests an update; g_i, client i gets a reply; gq_i and gu_i, the server gets a query and an update from client i; and sq_i and su_i, the server responds to a query and an update from client i. Table 5, in which i and j are distinct indices, gives the dependencies

	gu_i	gq_i	su_i	sq_i	gu_j	gq_j	su_j	sq_j	u_i	q_i	g_i	u_j	q_j	g_j
gu_i	d	i	d	i	d	d	i	i	d	i	i	i	i	i
gq_i	i	d	i	d	d	i	i	i	i	d	i	i	i	i
su_i	d	i	d	d	i	i	d	d	i	i	d	i	i	i
sq_i	i	d	d	d	i	i	d	d	i	i	d	i	i	i
u_i	d	i	i	i	i	i	i	i	d	d	d	i	i	i
q_i	i	d	i	i	i	i	i	i	d	d	d	i	i	i
g_i	i	i	d	d	i	i	i	i	d	d	d	i	i	i

Table 5: Dependencies Table for client-server protocol

among these transitions.

In checking the consistency property with the search command no explicit atomic propositions are defined. The search pattern, however, corresponds to a finite set of atomic propositions. The transitions that affect the satisfaction of these propositions are the ones that change auxiliary data, or the values the clients and server have. The remaining transitions, u_i, q_i, gu_i and gq_i, are the invisible transitions.

The basic idea presented here to take advantage of the POR approach is that the representation of the state may have two parts: one part allows the application of rules, the other one is frozen and does not. In [6] such a partitioning keeps a part of the state on which no transitions are enabled in the frozen part of the state. Here, to implement a POR technique, the frozen part of the state also includes parts of the state that have enabled transitions, but that are not in $ample(s)$.

Consider conditions $ample(s)$ must satisfy. Condition C2 requires that if $ample(s) \neq enabled(s)$ then every $\alpha \in ample(s)$ must be invisible. While one consequence of condition C1 is that all transitions in $enabled(s) \backslash ample(s)$ must be independent from those in $ample(s)$.

Consider whether $ample(s)$ could consist of the transitions of an individual client. Whenever `outstanding` is false and `request-count` $<$ `lim` for client i, invisible transitions u_i and q_i are enabled. Transitions dependent on these cannot be simultaneously enabled. Thus, any other enabled transitions would be independent from these. Now consider condition C1, which requires that for every path in the **full state graph** that starts at s some transition in $ample(s)$ must be executed before a transition dependent on one on $ample(s)$. The discarded paths start with prefixes of independent transitions. Let α be the first transition in these paths that is dependent on $\{u_i, g_i\}$. Suppose it is a transition of client i. Table 5 indicates that it could be one of these: u_i, q_i, g_i. The first two are in $ample(s)$, satisfying C1. For g_i to be enabled either u_i or q_i must have been executed before, but the transitions that were executed were independent of $\{u_i, g_i\}$, so this case is impossible. Now suppose α is one of the dependent server transitions: gu_i or gq_i. Again this case cannot arise since they must be preceded by u_i or q_i, respectively. This client-server specification has no cycles so condition C3 is satisfied. Thus, $ample(s)$ could be $\{u_i, g_i\}$. In such a state the entire configuration except client i would be in the frozen part of the state.

Further examination of Table 5 and the conditions $ample(s)$ must satisfy reveal that there is no $ample(s)$ that is a proper subset of $enabled(s)$ and contains transitions of the server.

As in the implementation of symmetry reduction, markers are used to indicate when a transition has occurred, and provide for an opportunity to discover $ample(s)$. Since the only $ample(s)$ is the one discussed above, a new marker is introduced: `op ! : NzNat -> Marker .`, which is used to indicate that u_i and g_i are enabled.

```
rl  [get-reply] :
    ? reply-to[ N : V' ]
    < client N | request-count : K, outstanding : true,  value : V >
 => < client N | request-count : K, outstanding : false, value : V' >
    !(N) .
```

There may be several clients that are enabled to send requests, so the representation of the state includes a list of the ids of those clients. Anytime the above transition is executed the id of the client is entered in the (ordered) list.

```
op _|*_ : Configuration Configuration -> Configuration
    [ctor frozen (2) format (n++i n--i n++i n--i)] .
op _#{_} : NzNatList Configuration -> TConfiguration
    [ctor frozen(1) format (n n n n++i n--i n)] .
op _#[_] : NzNatList Configuration -> TConfiguration [ctor frozen(1 2)] .
```

Only states that have a proper $ample(s)$ are partitioned in two, but after the transition is executed, a new ample set gets constructed.

```
eq (! C) |* C' = ! C C' .
eq (!(I) C) |* C' = !(I) C C' .
```

If client `I` has just been enabled to send requests, then `I` gets inserted in the list of ids.

```
eq L # {!(I) C} = (I ~> L) # [C] .
eq L # {! C} = L # [C]
```

The first id in the list determines the ample set.

```
eq nil # [C] = nil # {? C} .

ceq I, L # [ < client I |
                 request-count : N, outstanding : B, value : V >   C ]
 = L # { (? < client I |
                request-count : N, outstanding : B, value : V > ) |*  C }
if N < lim .

ceq I, L # [ < client I |
                 request-count : N, outstanding : B, value : V >   C ]
 = L # [ < client I |
             request-count : N, outstanding : B, value : V >   C ]
if N >= lim .
```

Once the ample set has been determined, and, if necessary, the state has been partitioned, the pretransition marker is reintroduced. Table 6 shows the result for the case with one server and two clients. The property was checked by three searches, one for each of the forms the state can take.

size	2
total states	8815
reduction	2.3%
cpu time (ms)	1321, 907, 895
real time (ms)	1332, 918, 903

Table 6: Metrics for the application of the POR technique.

7 Conclusion and Future Work

The design and development of highly assured distributed systems is challenged by the state-explosion problem, and often by the complexity of the properties to be verified. We developed two techniques for the reduction of the state space: an on-the-fly symmetry reduction, and an on-the-fly partial-order reduction. We also developed a technique for verifying strong, nontrivial properties that are not directly expressible in linear temporal logic. A couple of points are worth noting in appraising the value of this work.

First, in practice much of what is considered analysis of distributed systems is performed using simulations. These analyses examine only a few behaviors of the system. A Maude specification is executable, and so it can also be used for simulations by simply executing it. This may be used to perform some initial debugging of the specification or to gain a greater understanding of the system being developed. Simulation, however, whether formal or informal, is not verification, and is not a foundation for trusted systems. The techniques presented here support verification through exhaustive exploration of the state space. They contribute to a framework in which it is possible to document and specify, simulate and verify a system by modularly extending or transforming formal, object-oriented specifications. and thus they contribute to the advancement of formal support for the design of distributed systems.

Second, there is no single technique that will eliminate the state-explosion for all specifications. What are needed are techniques that can be composed and combined. The techniques developed in this study impose no restrictions on the specifications on which they are applied, and can be easily composed. They are implemented by simple transformations and modular extensions of the original specifications. No translation to different logics are needed.

Additional research needs to investigate general and protocol-specific optimizations to the two state-space reduction techniques that will reduce the time and memory overhead. Furthermore, in order to verify larger and more complex systems, composition of these and other techniques need to be studied, and applied to a collection a protocols, including some common benchmarks.

A new area of research is to explore notions and characterizations of data independence, which would permit the verification of large and infinite systems by the verification of much smaller systems.

References

[1] E. M. Clarke, O. Grumberg, D. Peled. Model Checking. MIT Press, Cambridge, MA, 2000.

[2] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science,* 285, 2002.

[3] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer. All About Maude — A High-Performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic. Springer Verlag, 2007.

[4] C. N. Ip, D. L. Dill. Better verification through symmetry. Formal Methods in System Design. V. 9 N.1–2, pages 41–75. August, 1996.1

[5] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[6] D. E. Rodríguez. On Modelling Sensor Networks in Maude. Electronic Notes in Theoretical Computer Science, Volume 176, pages 199–213, 2007.

[7] R. van Renesse, F. Schneider. Replication for Supporting High Throughput and Availability. *Proc. of the Sixth Symposium on Operating Systems Design and Implementation*, December 2004.